WARWICK
THE UNIVERSITY OF WARWICK

Manuscript version: Published Version
The version presented in WRAP is the published version (Version of Record).

Persistent WRAP URL:
http://wrap.warwick.ac.uk/134073

How to cite:
The repository item page linked to above, will contain details on accessing citation guidance from the publisher.

Copyright and reuse:
The Warwick Research Archive Portal (WRAP) makes this work of researchers of the University of Warwick available open access under the following conditions.

This article is made available under the Creative Commons Attribution 4.0 International license (CC BY 4.0) and may be reused according to the conditions of the license. For more details see: http://creativecommons.org/licenses/by/4.0/.

Publisher's statement:
Please refer to the repository item page, publisher's statement section, for further information.

For more information, please contact the WRAP Team at: wrap@warwick.ac.uk

warwick.ac.uk/lib-publications

Postcondition-Preserving Fusion of Postorder Tree Transformations

Eleanor Davies
University of Warwick
United Kingdom
Eleanor.Davies@warwick.ac.uk

Sara Kalvala
University of Warwick
United Kingdom
Sara.Kalvala@warwick.ac.uk

Abstract

Tree transformations are common in applications such as program rewriting in compilers. Using a series of simple transformations to build a more complex system can make the resulting software easier to understand, maintain, and reason about. Fusion strategies for combining such successive tree transformations promote this modularity, whilst mitigating the performance impact from increased numbers of tree traversals. However, it is important to ensure that fused transformations still perform their intended tasks. Existing approaches to fusing tree transformations tend to take an informal approach to soundness, or be too restrictive to consider the kind of transformations needed in a compiler. We use postconditions to define a more useful formal notion of successful fusion, namely postcondition-preserving fusion. We also present criteria that are sufficient to ensure postcondition-preservation and facilitate modular reasoning about the success of fusion.

CCS Concepts • **Software and its engineering** → **Compilers**; • **Theory of computation** → *Program reasoning*.

Keywords tree transformations, program rewriting, modular reasoning

ACM Reference Format:
Eleanor Davies and Sara Kalvala. 2020. Postcondition-Preserving Fusion of Postorder Tree Transformations. In *Proceedings of the 29th International Conference on Compiler Construction (CC '20), February 22–23, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3377555.3377884

1 Introduction

Consider a series of abstract syntax tree (AST) transformations in a compiler. An initial AST is created by parsing source code. Then, this AST is passed through a series of transformations, the output of each transformation becoming the input of the next. These transformations act to remove high-level language features or to optimise the underlying program. Finally, target code is generated from the transformed AST.

Ideally such AST transformations would be highly modular, with each performing a small, singular rewrite. However, the greater the number of separate transformations, the greater the number of AST traversals required. This can be detrimental to performance, especially when ASTs are large and each transformation only makes a slight change.

One approach to using transformations effectively is to automatically *fuse* transformations, maintaining modularity for the compiler developer, whilst reducing the number of AST traversals required when the compiler is used. To this end, Petrashko, Lhoták and Odersky [15] proposed and implemented miniphase fusion for the Dotty Scala compiler. Miniphases impose a structure on AST transforming compiler phases that allows them to be automatically fused. Analysis of miniphase fusion in Dotty demonstrated real benefits, to both modularity and performance.

To accommodate the complex nature of compiling Scala, there are no formal guarantees for the soundness of miniphase fusion. Instead, an informal set of guidelines as to when miniphases can be successfully fused is provided, relying on developer experience and detailed knowledge of the compiler. These guidelines are augmented by postcondition checks during testing, which ensure that fused miniphases still establish the invariants for which they were separately intended.

Relying solely on developer intuition and testing leaves room for problematic corner cases to slip through the net. In most work on fusing tree traversals, soundness is formally proved, yielding stronger correctness guarantees [9, 10, 16–19]. However, there are two factors that prevent the solutions proposed in such related work being directly useful in the case of miniphase-style fusion.

Firstly, the traversals being considered tend to involve limited transformations. In general, changes cannot be made to the children of a node, only to the data stored at the node itself. This is overly restrictive for compiler phases which need to make drastic structural AST changes.

Secondly, successful fusion is usually defined as producing a fused transformation that will always give the same result

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
CC '20, February 22–23, 2020, San Diego, CA, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7120-9/20/02.
https://doi.org/10.1145/3377555.3377884

as running its individual constituent transformations successively. However, we can demonstrate, with an example, that this precludes opportunities for fusion that would still be beneficial, particularly for AST optimising transformations.

We want to exploit fusion as much as possible, whilst still providing strong correctness guarantees. Therefore, we need a different formalisation approach, that can handle the kind of tree transformations required in a compiler, and that takes a wider perspective on when fusion is successful.

Key points. In this paper, we argue that preservation of postconditions, as exploited in Dotty, is a broader and more useful notion of successful fusion than preserving all observable behaviour of the tree transformations involved.

1. We illustrate, with an example, that requiring all behaviour to be preserved can prevent fusion that would actually be beneficial (Section 2).
2. Therefore, we present a wider definition of successful fusion, preserving postconditions that encapsulate the intentions of the programmer (Section 3).
3. We then derive and verify criteria for postcondition-preserving fusion, that allow modular reasoning at the level of individual tree transformations and their postconditions (Sections 3 and 4).
4. We also outline how these criteria can be checked, with techniques ranging from theorem proving to property-based testing (Section 5).

Additionally, Section 6 contains an overview of and comparison with related work on fusing tree transformations.

2 An Example

Suppose that we have a simple language consisting of binary arithmetic expressions and an `IF0` conditional construct. Then we can define an appropriate AST structure.

Definition 2.1. For the purposes of this section, we define a data type `Tree` as follows:

```
Tree := NAT Nat | BINOP Op Tree Tree | IF0 Tree Tree Tree
```

where `Op := ADD | SUB | MUL`, and `Nat` is a natural number type.

There are some rewrites that we could make to automatically optimise our ASTs.

Definition 2.2. If the children of a `BINOP` node are leaves, namely `NAT` nodes, we evaluate the expression and replace it with the result, using some sensibly defined `eval` function.

```
binop_eval (BINOP op (NAT n1) (NAT n2))
             := NAT (eval op n1 n2)
binop_eval t := t
```

Definition 2.3. We simplify `IF0` nodes, where the child representing the condition is `NAT 0`, so the first branch is taken.

```
zero_condition (IF0 (NAT 0) t1 t2) := t1
zero_condition t := t
```

To apply these rewrites effectively, we need a general function that traverses our ASTs and transforms them.

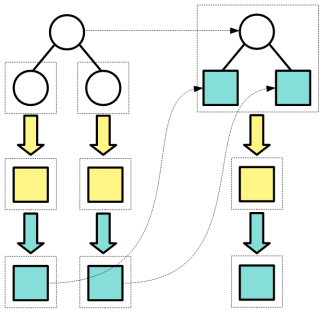

Figure 1. Applying two tree transformations in a single postorder traversal.

Definition 2.4. We define `transform` to perform a postorder `Tree` traversal, recursively applying a given rewrite function.

```
transform f (NAT n) := f (NAT n)
transform f (BINOP op t1 t2)
 := f (BINOP op (transform f t1) (transform f t2))
transform f (IF0 t1 t2 t3)
 := f (IF0 (transform f t1)
           (transform f t2) (transform f t3))
```

Hence, we can automatically optimise ASTs for our example language, by applying the tree transformations that we have defined. Currently, to apply both optimisations, we use `transform` to perform two separate `Tree` traversals:

```
transform zero_condition (transform binop_eval t).
```

Instead, we could combine the two transformations into a single traversal. As illustrated in Figure 1, for a given node, we first transform the node's children, using both fused transformations, before performing both rewrites on the node itself with its newly transformed children.

Definition 2.5. We define `fused` to take two rewrite functions and compose them such that, during a postorder `Tree` traversal, both rewrites are applied to each node visited.

```
fused f1 f2 t := transform (f2 ∘ f1) t
```

Thus, we can apply both optimisations to an AST within the same traversal, without having to manually combine the two transformations. For example, let `t0 :=`

```
BINOP ADD (NAT 1)
          (IF0 (BINOP MUL (NAT 1) (NAT 0)) (NAT 1) (NAT 0))
```

Then:

```
fused binop_eval zero_condition t0 = NAT 2
```

It would appear, at least for this particular `Tree`, that our fused optimisation attempt is successful: `binop_eval` has removed all `BINOP` nodes that could be immediately evaluated, `zero_condition` has removed all `IF0` nodes with a `NAT 0` condition, and the `Tree` has retained its inherent value or meaning.

In most related work, successful fusion must preserve the final result of a series of transformations. That is, for all `t`:

```
fused f1 f2 t = transform f2 (transform f1 t).
```

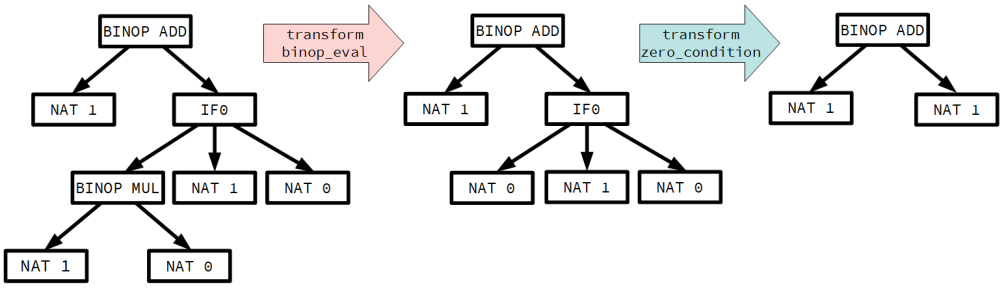

Figure 2. `binop_eval` and `zero_condition` applied separately in succession.

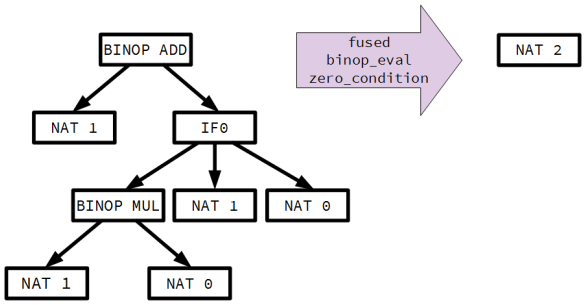

Figure 3. `binop_eval` and `zero_condition` applied fused together.

As illustrated in Figures 2 and 3, for our example `Tree t0` this does not hold, since:

```
transform zero_condition (transform binop_eval t0)
        = BINOP ADD (NAT 1) (NAT 1)
```

However, we find that fusing these two optimisations remains beneficial. In fact, being interleaved in this way has made our optimisations seemingly more effective. The fused transformations each uncover opportunities for the other to optimise further, whereas the result of applying them separately can be further optimised by `binop_eval`. So, we seek a more permissive view of when fusion is deemed successful.

3 Criteria for Successful Fusion

"The most important property of a program is whether it accomplishes the intention of its user." - C. A. R. Hoare [8]

3.1 What Do We Mean By "Successful" Fusion?

Tree transformations, such as AST transforming compiler phases, are inevitably written to perform a given job. For instance, our example transformations in the previous section were intended to optimise away given syntactic patterns. It is therefore vital that, if separately run tree transformations accomplish their intended behaviours, then so does the result of fusing them.

To preserve the intended behaviour of tree transformations, under fusion, it is sufficient to preserve all observable behaviour. A common interpretation of successful fusion is that the observable behaviour of fused transformations should be identical to that of the same transformations running individually one after the other. However, the previous example demonstrates that there are cases which this does not capture. We are perhaps being overly conservative in trying to preserve behaviour that was never necessarily wanted.

It is not always possible to accurately determine the intentions of a developer from the code that they produce. In Dotty miniphases [15] postconditions are implemented by compiler developers, to encode the intended behaviour of a tree transformation. Checks during testing then ensure that they are preserved by fusion. Analogously, we will think of "successful" fusion in terms of fusion that preserves relevant postconditions, as defined informally below.

Definition 3.1. Let `f1` and `f2` be tree transformations, each fulfilling a given postcondition, `p1` and `p2` respectively. We say that `f1` and `f2` can be successfully fused, with respect to `p1` and `p2`, if `fused f1 f2` also fulfils both `p1` and `p2`. We refer to this as *postcondition-preserving fusion*.

This definition parameterises the success of fusion over specific postconditions. Thus, we ensure that the intended behaviour of our tree transformations will be preserved. Moreover, the postconditions specify exactly what the intended behaviour is. If a given postcondition poses an obstacle to fusing transformations, it may be that the developer can refine their expectations and write a new, less ambitious, postcondition.

Here we will consider purely functional tree transformations, that is functions that take a tree and return another

tree, without any side-effects. Hence, we need only look at the inputs and outputs of our transformations, when examining behaviours. This means that postconditions can be specified as predicates on trees.

3.2 Criteria For Postcondition-Preserving Fusion

Now that we have our notion of success as postcondition-preservation, we need to know when fusion will be successful. It is possible, as in Dotty, to wait until tree transformations are fused and check that the postconditions are fulfilled. However, this means testing or reasoning about the resulting complex fused transformation, which conflicts with our original goal of modularity.

Instead, we propose reasoning about the individual tree transformations before they are fused. Therefore, we need criteria to tell us that, given some transformations and corresponding postconditions, fusing them will be postcondition-preserving. In this section we develop and justify such criteria, with reference to our previous example.

First we define postconditions for our optimising transformations. In this case, we simply require that the rewritten AST should contain no instances of the respective syntactic pattern that we intended to optimise away.

Definition 3.2. For `binop_eval` we have the postcondition:

```
post_binop_eval (BINOP op (NAT n1) (NAT n2)) := False
post_binop_eval t := True
```

For `zero_condition` we have the postcondition:

```
post_zero_condition (IF0 (NAT 0) t1 t2) := False
post_zero_condition t := True
```

Suppose we apply `fused binop_eval zero_condition` to a `Tree`, and require that postconditions `post_binop_eval` and `post_zero_condition` hold afterwards.

Criterion 1. As we visit a node, on our postorder traversal, we first rewrite it with `binop_eval`. If we have defined our transformation and postcondition correctly, then we know that `post_binop_eval` should hold after this rewrite. Next we have to rewrite the node with `zero_condition`. To be sure that `post_binop_eval` will still hold after this rewrite, we need to know that `zero_condition` preserves it.

Criterion 2. When applying `zero_condition` individually, any children of a node that it rewrites have just been rewritten by `zero_condition` themselves, so we can assume that they already satisfy `post_zero_condition`. If we are applying `fused binop_eval zero_condition` instead, the node will first be rewritten by `binop_eval`. Therefore, if `binop_eval` makes any changes to the node's children, they may no longer satisfy `post_zero_condition`. So, we need to know that `binop_eval` preserves `post_zero_condition` for all children of the nodes that it rewrites.

3.3 Formalising and Verifying Our Criteria

Here we outline how we have formalised and verified our criteria, generalising from our example. We have also mechanised work from this section and Section 4, using the Coq proof assistant [22]. [1]

We parameterise our definition of trees over a data type `X` representing node labels. So in the case of our example, `X := NAT Nat | BINOP Op | IF0`. To highlight the inductive nature of the following definitions, we differentiate between a `Leaf` and an inner `Node`, although this is not strictly necessary as a `Leaf` is just a `Node` with an empty list of children.

Definition 3.3. We define a `Tree` as: a `Leaf` labelled from some set of labels, or a `Node` with a label and a child list.

```
Tree := Leaf X | Node X (List Tree)
```

for some label type `X`.

As in our example, we separate our rewrite rules from the process of traversing and transforming the tree. This allows us to impose a standardised postorder traversal, and hence automatically fuse our transformations.

Definition 3.4. We define a function `transform` that takes a rewrite function `f : Tree → Tree` and applies it recursively to a given `Tree`.

```
transform f (Leaf x) := f (Leaf x)
transform f (Node x cs)
          := f (Node x (map (transform f) cs))
```

Our definition of fusion is then identical to that in Definition 2.5, taking two rewrite functions and applying both to each node visited, during a postorder `Tree` traversal.

Definition 3.5. For rewrite functions `f1, f2 : Tree → Tree` and `Tree t`, we define `fused` as:

```
fused f1 f2 t := transform (f2 ∘ f1) t
```

where ∘ is standard function composition.

Now we can think about postconditions for our transformations. We define postconditions as predicates over `Trees`, allowing us to express some `Tree` property that must hold after a tree has been transformed. Particularly with AST transformations in compilers, we would commonly want such a property to hold for the entire `Tree`. Hence, we define a function to check recursively that some predicate holds for a node and all of its descendants.

Definition 3.6. For a predicate `p : Tree → {True, False}`, we define a function to check it recursively:

```
check p (Leaf x) := p (Leaf x)
check p (Node x cs) := p (Node x cs)
                     ∧ ∀ c ∈ cs, check p c
```

We do not concern ourselves here with whether a postcondition is appropriate for the tree transformation it describes. If a transformation is not behaving as expected before fusion,

[1] https://github.com/EleanorRD/postcondition-preserving-fusion

then its behaviour after fusion is largely irrelevant. Therefore we will need to assume that a tree transformation satisfies its defined postcondition. It is then the developer's job to convince themselves of this.

Given that all of our rewrite rules are applied recursively, we can assume that all descendants of a node being rewritten have already been rewritten and hence already fulfil the postcondition. Therefore, we say that a rewrite function `f` `satisfies` a postcondition `p` if rewriting any node, whose children fulfil `p`, will result in a `Tree` that fulfils `p`.

Definition 3.7. For some rewrite function `f` and postcondition `p`, we define `satisfies` as:

```
satisfies f p := ∀ t, (∀ c ∈ children t, check p c)
                    ⇒ check p (f t)
```

Having established that postconditions are appropriate for their respective tree transformations, we define the necessary relationships between a transformation and the postcondition of the other transformation that it is being fused with. This relationship is not symmetric, differing depending on whether the transformation is the first or second of the fused pair. As explored in our example, our fusion criteria for postcondition-preservation are as follows.

Definition 3.8. For some rewrite function `f` and postcondition `p`, we have two criteria:

```
FC1 f p := ∀ t, check p t ⇒ check p (f t)
FC2 f p := ∀ t, (∀ c ∈ children t, check p c)
             ⇒ ∀ c′ ∈ children (f t), check p c′
```

These definitions can then be used to express criteria that ensure that fusing two tree transformations will preserve given corresponding postconditions. Suppose we are fusing `f1` and `f2`, in that order, with postcondition `p1` and `p2` respectively. We require `FC1 f2 p1` so that we know `f2` preserves `p1`. We also require `FC2 f1 p2` so that we know that `f1` preserves `p2` in any children of the node that it is rewriting.

Therefore, with our criteria, we can assemble our theorem for postcondition-preserving fusion of tree transformations.

Theorem 3.9. *Let* `f1` *and* `f2` *be tree rewrite functions, and* `p1` *and* `p2` *be postcondition predicates. Then:*

```
satisfies f1 p1 ∧ satisfies f2 p2 ∧ FC1 f2 p1 ∧ FC2 f1 p2
          ⇒ ∀ t, check p1 (fused f1 f2 t)
                ∧ check p2 (fused f1 f2 t)
```

In proving this theorem, and hence verifying our fusion criteria, we can split it into two lemmas, each considering a different postcondition.

Lemma 3.10. *Let* `f1` *and* `f2` *be rewrite functions and* `p1` *a postcondition. Then:*

```
satisfies f1 p1 ∧ FC1 f2 p1
          ⇒ ∀ t, check p1 (fused f1 f2 t)
```

Proof. We prove this lemma by induction, on the structure of the `Tree` data type.

Let `t = Leaf x`. Then `fused f1 f2 t` becomes `f2 (f1 t)`. We know that `f1 t` fulfils `check p1`, due to `satisfies f1 p1`. Hence, `f2 (f1 t)` also fulfils `check p1`, as `FC1` requires that `f2` preserves `p1`.

Now, let `t = Node x cs`. Then `fused f1 f2 t` becomes `f2 (f1 (Node x (map (fused f1 f2) cs)))`. Our induction hypothesis is that `check p1 (fused f1 f2 c)`, for all `c` in `cs`. Therefore, `f1 (Node x (map (fused f1 f2) cs))` fulfils `check p1`, due to `satisfies f1 p1`. And so, as before, `FC1` ensures that `f2 (f1 (Node x (map (fused f1 f2) cs)))` also fulfils `check p1`. □

Lemma 3.11. *Let* `f1` *and* `f2` *be rewrite functions and* `p2` *a postcondition. Then:*

```
satisfies f2 p2 ∧ FC2 f1 p2
          ⇒ ∀ t, check p2 (fused f1 f2 t)
```

Proof. As for the previous lemma, we proceed by induction.

Let `t = Leaf x`. Then `fused f1 f2 t` becomes `f2 (f1 t)`. Since `t` has no children, `FC2` dictates that any children of `f1 t` must fulfil `check p2`. Therefore, `f2 (f1 t)` fulfils `check p2`, due to `satisfies f2 p2`.

Now, let `t = Node x cs`. Then `fused f1 f2 t` becomes `f2 (f1 (Node x (map (fused f1 f2) cs)))`. Our induction hypothesis is that `check p2 (fused f1 f2 c)`, for all `c` in `cs`. So, any children of `f1 (Node x (map (fused f1 f2) cs))` must fulfil `check p2`, due to `FC2`. Therefore, we know that `f2 (f1 (Node x (map (fused f1 f2) cs)))` fulfils `check p2`, since we know `satisfies f2 p2`. □

4 From Pairwise To Multiple Fusion

In order to fully reap the benefits of fusion, we want to be able to fuse a list of multiple transformations. Hence, in this section we extend our pairwise results to consider a list of arbitrary-many tree transformations to fuse.

We define `compose_list`, which recursively composes a list of tree rewrite functions, and which we can use in place of `compose` in our definition `fused_list`.

Definition 4.1. For a list of rewrite functions and a `Tree t`, we define:

```
compose_list [] t := t
compose_list (f :: fs) t := compose_list fs (f t)
```

Definition 4.2. For some list of rewrite functions `fs` and `Tree t`, we define:

```
fused_list fs t := transform (compose_list fs) t
```

Now that we can fuse multiple transformations together, we need to consider how our fusion criteria will scale. We choose an arbitrary rewrite function `f`, from our list, with its corresponding postcondition `p`. Using `FC1` and `FC2`, we can split our list into three parts `before ++ [f] ++ after` and derive separate criteria for the transformations before and after `f` in the fusion order.

A transformation after f must preserve p so as not to undo the work that f has done. So all transformations after f must fulfil FC1 with respect to p. A transformation before f must not introduce children that violate p to ensure that f can fulfil p. So all transformations before f must fulfil FC2 with respect to p. Hence we define below our extended criteria.

Definition 4.3. For some list of rewrite functions fs and postcondition p, we have two fusion criteria:

```
after_FC1  p fs := ∀ f ∈ fs, FC1 f p
before_FC2 p fs := ∀ f ∈ fs, FC2 f p
```

Thus, for any given transformation in our list, if it satisfies its corresponding postcondition, and both extended fusion criteria are satisfied, then the result of applying our fused list will fulfil that postcondition.

Theorem 4.4. *Let* gs ++ [f] ++ hs *be a list of rewrite functions and* p *a postcondition. Then:*

```
satisfies f p ∧ before_FC2 p gs ∧ after_FC1 p hs
⇒ ∀ t, check p (fused_list (gs ++ [f] ++ hs) t)
```

The proof of this theorem proceeds by nested induction on various parts of the list, and on the Tree data type structure. We split this into a number of lemmas that we will prove before returning to the proof of Theorem 4.4.

Lemma 4.5. *Let* t *be a* Tree, p *a postcondition, and* fs *a list of rewrite rules. Then:*

```
check p t ∧ after_FC1 p fs
          ⇒ check p (compose_list fs t)
```

Proof. We prove this lemma by induction on fs. If fs is an empty list then compose_list fs t is just t, and we are done.

Let fs = f :: fs′. Then compose_list fs t becomes compose_list fs′ (f t). Our induction hypothesis is, for any Tree t′, check p t′ ⇒ check p (compose_list fs′ t′). Hence, we just need to prove check p (f t), which directly follows from after_FC1 p (f :: fs′). □

Lemma 4.6. *Let* [f] ++ hs *be a list of rewrite functions and* p *a postcondition. Then:*

```
satisfies f p ∧ after_FC1 p hs
    ⇒ ∀ t, check p (fused_list ([f] ++ hs) t)
```

Proof. We begin by induction on the list hs.

Let hs be an empty list. Then fused_list ([f] ++ hs) t becomes transform f t. We proceed by induction on t. If t = Leaf x, then transform f t is just f t, and check p (f t) follows from satisfies f p, as t has no children.

Otherwise, we have t = Node x cs, and so transform f t becomes f (Node x (map (transform f) cs)). Our induction hypothesis is that, for all c in cs, check p (transform f c). So, we have check p (f (Node x (map (transform f) cs))), again from satisfies f p.

Now, let hs = hs′ ++ [h]. Our induction hypothesis is, for any Tree t′, check p (fused_list ([f] ++ hs′) t′). We proceed by induction on t.

Let t = Leaf x. Then fused_list ([f] ++ hs) t becomes h (compose_list hs′ (f t)). Due to our hs induction hypothesis, we know that check p (compose_list hs′ (f t)) holds. From after_FC1 p hs we have FC1 h p. Therefore, we have check p (h (compose_list hs′ (f t))).

Now, let t = Node x cs. Then fused_list ([f] ++ hs) t becomes h (compose_list hs′ (f (Node x (map (fused_list ([f] ++ hs)) cs)))). Our induction hypothesis for t is that check p (fused_list ([f] ++ hs) c) holds, for all c in cs. So, check p (f (Node x (map (fused_list ([f]++hs)) cs))) follows from satisfies f p. Thus, from Lemma 4.5, we have check p (compose_list hs′ (f (Node x (map (fused_list ([f]++hs)) cs)))). Finally, from after_FC1 p hs we have FC1 h p, and so h preserves check p and we are done. □

Lemma 4.7. *Let* t *be a* Tree, p *a postcondition, and* fs *a list of rewrite rules. Then:*

```
(∀ c ∈ children t, check p c) ∧ before_FC2 p fs
⇒ ∀ c′ ∈ children (compose_list fs t), check p c′
```

Proof. We prove this lemma by induction on fs. If fs is an empty list then compose_list fs t is just t, and we are done.

Let fs = f :: fs′. Then compose_list fs t becomes compose_list fs′ (f t). Our induction hypothesis is, for any Tree t′, we have (∀ c ∈ children t′, check p c) ⇒ ∀ c′ ∈ children (compose_list fs′ t′), check p c′. Hence, we just need to show ∀ c ∈ children (f t), check p c, which follows directly from before_FC2 p (f :: fs′). □

Proof of Theorem 4.4. Let gs ++ [f] ++ hs be a list of rewrite functions and p a postcondition. Suppose: satisfies f p and before_FC2 p gs and after_FC1 p hs. We want to show that:

```
∀ t, check p (fused_list (gs ++ [f] ++ hs) t).
```

We begin by case analysis on the list gs. If gs is the empty list, then we are just considering functions [f] ++ hs, which is exactly the case covered by Lemma 4.6.

Instead, let gs = g :: gs′. We proceed by induction on t.

Let t = Leaf x. Then fused_list (gs ++ [f] ++ hs) t becomes compose_list hs (f (compose_list gs′ (g t))). Since t is a Leaf, and consequently has no children, we can say that ∀ c ∈ children t, check p t. Therefore, we have ∀ c ∈ children (compose_list gs′ (g t)), check p t, due to Lemma 4.7 and before_FC2 p gs. From satisfies f p, we then have check p (f (compose_list gs′ (g t))). And, check p (compose_list hs (f (compose_list gs′ (g t)))) follows from Lemma 4.5.

Let t = Node x cs. Then fused_list (gs++[f]++hs) t becomes compose_list hs (f (compose_list gs′ (g t′))), where t′ = Node x (map (fused_list (gs++[f]++hs)) cs). Our induction hypothesis is that, for all c in cs, we know check p (fused_list (gs ++ [f] ++ hs) c) holds. Thus, we can follow the same reasoning as the Leaf case to complete the proof. □

5 Making Use Of Our Criteria

The main reason behind developing a set of formal criteria for postcondition-preserving fusion is to allow modular reasoning. We can think about `f1` and `f2` individually, rather than having to determine whether `fused f1 f2 t` will always fulfil the relevant postconditions. Namely, for pairwise fusion, we are checking:

`satisfies f1 p1`, `satisfies f2 p2`, `FC1 f2 p1` and `FC2 f1 p2`, rather than:

$\forall$ `t`, `check p1 (fused f1 f2 t)` $\wedge$ `check p2 (fused f1 f2 t)`.

In this section, we explore a range of techniques that could be used to check the criteria prior to fusion. Each of these approaches has its benefits and may be suited to different situations. Modularity is beneficial in each of these cases.

5.1 Using a Proof Assistant

Proof assistants, or interactive theorem provers, allow developers to implement software and formally prove that it satisfies desired properties, such as our fusion criteria. Having simple modular tree transformations to reason about will make this process easier for the inexperienced user, as there is generally a steep learning curve in using proof assistants. Moreover, if the proofs are reasonably simple, there is more chance that the small amounts of proof automation, present in such tools, will be helpful.

The Coq proof assistant [22] is a popular choice for verifying software. We have used Coq to mechanise and check the formalisation and proofs in this paper. This not only substantiates the work, but also leaves behind a framework that could be used as a template to implement specific tree transformations and prove that the fusion criteria hold.

Verified code implemented in Coq can be directly extracted into a number of languages, such as OCaml and Haskell. Hence, the software does not need to be implemented a second time for verification. There are also projects like `hs-to-coq` [1, 20], which aims to automatically translate Haskell code into Gallina, the Coq specification language. Such tools help to bridge the gap between verification, via theorem proving, and software implementation.

5.2 Automated Static Analysis

Another, more automated, possibility is to use some form of static code analysis software. Static analysis examines the behaviour of a program without executing it, and can exploit tools like SMT solvers to check that given properties hold. Such approaches tend to be less expressive than using a proof assistant, but provide the benefit of automation.

With automated verification methods, modularity is valuable. Having smaller components and simpler properties to check helps to avoid issues such as state-explosion. Moreover, many such methods search for a counterexample input, if a proof cannot be found. A counterexample is most useful if it can be directly linked with a specific section of code, to effectively diagnose the problem. This is more likely to happen if the verification effort is highly modular.

5.3 Runtime Checks

Contract syntax is often used to express properties to be checked at runtime. This can exist in native form in a language, such as JML, or as part of a library, such as `Predef` in Scala. Keywords like `require` and `ensuring` are used to express preconditions and postconditions at a given point in code. Exceptions may be raised if these conditions are not met. Contracts can be permanently in place during runtime or toggled for use only during testing.

If we are implementing modular tree transformations, it is necessary for the conditions that we are checking to also be modular. It is not straightforward to express, using contracts, a property referring to multiple tree transformations which are defined in different parts of code. Therefore, our criteria are useful here, in that each only relates to one tree transformation. Postconditions can then be expressed as part of the contract syntax.

5.4 Property-Based Testing

Property-based testing frameworks are designed to test for specific properties, using a large number of appropriate random inputs. This can be used in a similar way to contract syntax, during testing, however it removes the required properties from the code itself. It also allows the user to express properties about multiple elements, rather than just one point in the code.

The advantage of modularity here, is that the tests can be very specific. If a given test fails, it is easy to localise which criteria was not met, as well as which particular tree transformation caused the problem.

5.5 Generalising Proofs For Removal Rewrites

It is also possible to prove that rewrite rules will always fulfil the criteria, if they have some other property which is even easier to check. Making generalisations like this can further reduce the effort required in verification or testing. For example, in this section we prove that a rewrite function which just acts to remove a section of the tree will satisfy our fusion criteria with respect to any postcondition.

We will call a rewrite function a removal if rewriting a tree always returns either the tree itself or one of its children. Hence, the transformation is simply removing part of the tree, if it changes the tree at all. This is the case for the example transformation `zero_condition`, that we defined in Section 2, and for other simple AST optimisations such as folding Boolean expressions.

Definition 5.1. A rewrite function `f` is a *removal* if, for every `Tree t`, either `f t = t` or `f t` $\in$ `children t`. We will denote this property as: `removal f`.

We can prove that all removal transformations will satisfy both pairwise fusion criteria `FC1` and `FC2`, for all possible postconditions.

Lemma 5.2. *Let* `f` *be a rewrite function and* `p` *a postcondition. Then:* `removal f ⇒ FC1 f p`.

Proof. Let `t` be a `Tree` and suppose that `check p t` holds. We need to show that `check p (f t)` holds. We proceed by case analysis on `t`.

Let `t = Leaf x`. Due to `removal f`, we know that `f t` is either just `t` or a child of `t`. Since `t` is a `Leaf` and thus has no children, it must be the case that `f t = t`. And we already know that `check p t` holds.

Now, let `t = Node x cs`. If `f t = t`, then as before, we are done. Otherwise, `f t ∈ children t`. Due to the recursive nature of `check`, we already know that `check p c` holds for any `c ∈ children t`. □

Lemma 5.3. *Let* `f` *be a rewrite function and* `p` *a postcondition. Then:* `removal f ⇒ FC2 f p`.

Proof. Let `t` be a `Tree` and suppose that `∀ c ∈ children t, check p c`. We show `∀ c′ ∈ children (f t), check p c′`, proceeding by case analysis on `t`.

Let `t = Leaf x`. As in the previous lemma, since `t` has no children, it must be the case that `f t = t`. And we have already assumed that `∀ c ∈ children t, check p c`.

Now, let `t = Node x cs`. Again, if `f t = t`, we are done. Otherwise, `f t ∈ children t` and the recursive definition of `check` grants us what we seek. □

Checking that `removal f` holds should be straightforward and directly related to the implementation of any given rewrite function. By generalising our proofs, in this way, we can yield strong guarantees without having to write very similar proofs for similar tree transformations.

6 Related Work

6.1 Fusing Compiler Phases

Our work is largely inspired by the miniphase approach implemented in the Dotty compiler [14, 15]. Most commonly, Scala is compiled to Java bytecode, by generating an abstract syntax tree (AST) which is transformed multiple times before being used to generate appropriate bytecode. These AST transformations rewrite high-level features using lower-level concepts and optimise program code. Dotty provides a template for implementing AST transformations, or miniphases, which allows them to be automatically fused. This includes imposing a postorder AST traversal.

Dotty illustrates that useful tree transformations can still be implemented within a template like this. However, not all miniphases are fused, with that section of the compiler consisting of several distinct fused blocks. Petrashko et al. [15] give a series of high-level criteria which determine whether miniphases are fusible. One criterion involves preserving invariants, which is ensured in practice by dynamic postcondition checks for each miniphase during testing.

Thus, the concept of postcondition-preserving fusion is central to considering successful fusion in Dotty. Petrashko et al. argue that their fusion criteria are more easily applicable to realistic tree transformations than strict soundness criteria. Further to this, we have demonstrated that, even for some very simple tree transformations, strict soundness is overly restrictive, and that fusion can make certain transformations more effective.

Through implementing miniphases in Dotty, Petrashko et al. establish the real-world benefits of such a fusion approach. Empirically, they show improvements in compiler running time and overall memory usage. Miniphase fusion results in a 35% reduction in the time taken by their tree transformations. Anecdotally, they also detail advantages to the open-source development approach, in making it easier for multiple programmers to understand the codebase and contribute towards independent compiler phases.

There are two features of miniphases that our work does not currently consider, but could be extended to incorporate. Firstly, transformations in miniphases can examine and alter entities outside of the AST being transformed. Secondly, in addition to postconditions, miniphases may have preconditions in the form of a list of other miniphases that must have already been run. These are both aspects that we could consider incorporating in the future.

6.2 Deforestation and Stream Fusion

The miniphase framework itself builds on the idea of deforestation [2, 6, 23]. Particularly in functional programming, developers build up complex functions by successively applying simpler functions, which often communicate through intermediate data structures. Deforestation aims to avoid explicitly generating these intermediate data structures by automatically fusing the constituent functions.

As initially proposed by Wadler [23], deforestation deals with fusing a very restrictive set of functions. For instance, functions must be in treeless form so they cannot construct any internal data structures themselves. Efforts to extend deforestation to a wider range of functions often involve identifying functions which cannot be fused and abstracting them out of the process.

Shortcut deforestation [6] standardises the way that lists are produced and consumed, using `foldr/build` pairs which can be cancelled. This has evolved into stream fusion [3, 12] which applies to a wider range of intermediate data structures and is used in the Glasgow Haskell Compiler.

Some approaches use category theory to reason about deforestation and fusion [7]. Takano and Meijer [21] extended shortcut deforestation beyond lists, to other data structures. Generalising the idea of `foldr/build` pairs to hylomorphisms, existing results on the fusion of hylomorphisms [13] can be applied to the case of deforestation. Nistal et al. implemented

a similar categorical approach to fusion in Coq [5], which we could have exploited in our Coq developments. However, for practical applications where developers should interact with the framework, it can be more approachable to maintain a syntactic perspective.

Tree transformations can also be modelled using tree transducers. For instance, Fast [4] is a language based on tree transducers to examine programs involving functions over trees. Moreover, Jürgensen and Vogler [11] showed that syntactic composition of top-down tree transducers is equivalent to shortcut fusion. The transformations in our work are inherently different as requiring a postorder traversal imposes a bottom-up approach instead.

6.3 Fusing Tree Traversals

There is a significant amount of existing work that focuses on fusing tree traversals, in which a tree is traversed multiple times to compute some final values. Generally, these approaches allow traversals to alter the information stored at a node but not the children of a node. This allows for stronger correctness guarantees, but limits the kind of tree transformations that can be specified.

Temporal locality can be exploited to improve the performance of tree traversals. By successively running the traversals which visit the same nodes, the node data will still be in cache, making it easier to retrieve. Given a series of points that traverse and interact with a tree, point blocking [9] involves sorting these points into blocks, depending on the nodes that they visit. Each block performs a single tree traversal and, at each node, all points in the block that interact with that node are applied. Point blocking relies heavily on preprocessing to sort the points into appropriate blocks. Traversal splicing [10] is a similar technique that sorts points dynamically, as they are being applied, to enhance locality.

The work of Jo and Kulkarni [9, 10] on enhancing locality focused mainly on independent tree traversals. Weijiang et al. [24] developed a static dependence test to extend these techniques to a wider range of traversals, that may interact with each other. In analysing the node access path in the traversal algorithms, the test determines whether point blocking or traversal splicing could be applied safely and when node visits can be reordered.

Rajbhandari et al. [17, 18] looked at automatically finding the optimal fusion schedule for recursive traversals of k-d trees. In particular, they considered the MADNESS system, which is designed for numerical scientific simulations. They examined the data dependencies of traversals based on their consumer-producer relationships and showed that fusing operators by interleaving them can improve performance by improving locality, as the trees used are often larger than cache.

TreeFuser [19] is a framework that looks to automatically fuse more general tree traversals. It employs code motion and partial fusion to perform as much fusion as possible. Code motion involves rearranging code such that fusion becomes feasible, for instance by changing the traversal order. Partial fusion considers traversals that cannot be fused completely, but can be fused over parts of the tree, still improving performance. TreeFuser produces a dependence graph that is used to determine when these techniques are applicable.

Qiu and Wang [16] implemented a decidable fragment of the Dryad logic for reasoning about trees. $\text{Dryad}_{\text{dec}}$ is especially suited to analysing tree traversals which calculate some measurement of the tree. One sample $\text{Dryad}_{\text{dec}}$ application is to check whether the fusion of a certain set of tree traversals is allowed, that is whether the fused traversals will have identical behaviour.

6.4 AST Transformations and Future Work

The tree transformations that we have considered, as examples in this paper, perform simple AST rewrites generally used in preprocessing compiler steps. More involved tree transformations tend to occur in later compiler phases, for example instruction selection by maximal munch or dynamic programming techniques. These would be interesting case studies on which to further evaluate our results.

As our initial source of inspiration, Dotty [15] provides a wealth of compiler phases, with which to assess the applicability of our theoretical work to realistic transformations. The AST transformations in Dotty are performed using a postorder traversal and hence inherently suited to our definitions. Although as mentioned before, since the transformations are not purely functional, we would have to either alter them or extend our formalisation.

7 Conclusion

Ordinarily the increased number of transformations promoted by modularity would result in an increased number of tree traversals and, hence, worse performance. Automatically fusing tree transformations can avoid this trade-off of modularity against performance. Fusion allows multiple transformations to be performed in a single traversal, mitigating the performance impact. Such a strategy has been successfully adopted for AST transformations as miniphases in the Dotty Scala compiler, implemented by Petrashko et al. [15].

A crucial consideration, when fusing tree transformations, is correctness. Fused transformations must continue to be useful, by performing their intended task. Most work on fusing tree traversals or transformations deems fusion successful if the fused transformations will produce an identical outcome to the same separate transformations run consecutively. However, this precludes some fusion opportunities that would still be beneficial. Moreover, many related techniques focus on a highly restricted set of tree transformations, in order to prove such soundness guarantees.

Instead, we have argued for a broader notion of what we mean by "successful", namely postcondition-preserving fusion. We use postconditions to encode the required behaviours of a tree transformation, allowing the developer of the transformation to specify what particular behaviour is important to them. We can then reason about whether that behaviour is preserved, rather than trying to preserve behaviour that is merely coincidental.

We have also derived and verified criteria that are sufficient to guarantee that a given set of tree transformations can be successfully fused, with respect to a given set of postconditions. Instead of reasoning about the final fused transformation, we are able to reason about the tree transformations individually. This will allow modular verification or testing which appropriately complements the modularity of the implemented tree transformations.

Acknowledgments

Eleanor Davies is funded by a studentship from the Engineering and Physical Sciences Research Council (EPSRC) under grant number EP/N509796/1.

References

[1] Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. 2018. Ready, set, verify! Applying hs-to-coq to real-world Haskell code (Experience report). *Proceedings of the ACM on Programming Languages* 2, ICFP (July 2018), 89. https://doi.org/10.1145/3236784

[2] Wei-Ngan Chin. 1992. Safe fusion of functional expressions. *ACM SIGPLAN Lisp Pointers* V, 1 (January 1992), 11–20. https://doi.org/10.1145/141478.141494

[3] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream Fusion: From lists to streams to nothing at all. *ACM SIGPLAN Notices* 42, 9 (October 2007), 315–326. https://doi.org/10.1145/1291151.1291199

[4] Loris D'Antoni, Margus Veanes, Benjamin Livshits, and David Molnar. 2015. Fast: A transducer-based language for tree manipulation. *ACM Transactions on Programming Languages and Systems* 38, 1 (October 2015), 1. https://doi.org/10.1145/2791292

[5] José L. Freire Nistal, José E. Freire Brañas, Antonio Blanco Ferro, and Juan J. Sánchez Penas. 2001. Fusion in Coq. In *International Conference on Computer Aided Systems Theory*. Springer, Berlin, Heidelberg, 583–596.

[6] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. ACM, New York, 223–232. https://doi.org/10.1145/165180.165214

[7] Ralf Hinze, Thomas Harper, and Daniel W. H. James. 2011. Theory and practice of fusion. In *Implementation and Application of Functional Languages (Lecture Notes in Computer Science, Vol. 6647)*. Springer, Berlin, Heidelberg, 19–37. https://doi.org/10.1007/978-3-642-24276-2_2

[8] Charles A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (October 1969), 576–580.

[9] Youngjoon Jo and Milind Kulkarni. 2011. Enhancing locality for recursive traversals of recursive structures. *ACM SIGPLAN Notices* 46, 10 (October 2011), 463–482. https://doi.org/10.1145/2048066.2048104

[10] Youngjoon Jo and Milind Kulkarni. 2012. Automatically enhancing locality for tree traversals with traversal splicing. *ACM SIGPLAN Notices* 47, 10 (October 2012), 355–374. https://doi.org/10.1145/2384616.2384643

[11] Claus Jürgensen and Heiko Vogler. 2004. Syntactic composition of top-down tree transducers is short cut fusion. *Mathematical Structures in Computer Science* 14, 2 (April 2004), 215–282. https://doi.org/10.1017/S0960129503004109

[12] Geoffrey Mainland, Roman Leshchinskiy, and Simon L. Peyton Jones. 2017. Exploiting vector instructions with generalized stream fusion. *Commun. ACM* 60, 5 (April 2017), 83–91. https://doi.org/10.1145/3060597

[13] Erik Meijer, Maartan Fokkinga, and Ross Paterson. 1991. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conference on Functional Programming Languages and Computer Architecture (Lecture Notes in Computer Science, Vol. 523)*. Springer, Berlin, Heidelberg, 124–144. https://doi.org/10.1007/3540543961_7

[14] Dmytro Petrashko. 2017. *Design and implementation of an optimizing type-centric compiler for a high-level language.* PhD Thesis. EPFL.

[15] Dmitry Petrashko, Ondřej Lhoták, and Martin Odersky. 2017. Miniphases: compilation using modular and efficient tree transformations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 201–216. https://doi.org/10.1145/3140587.3062346

[16] Xiaokang Qiu and Yanjun Wang. 2019. A Decidable Logic for Tree Data-Structures with Measurements.. In *International Conference on Verification, Model Checking, and Abstract Interpretation (Lecture Notes in Computer Science, Vol. 11388)*. Springer, Cham, 318–341. https://doi.org/10.1007/978-3-030-11245-5_15

[17] Samyam Rajbhandari, Jinsung Kim, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, Robert J. Harrison, and Ponnuswamy Sadayappan. 2016. A domain-specific compiler for a parallel multiresolution adaptive numerical simulation environment.. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, Salt Lake City, 40. https://doi.org/10.1109/sc.2016.39

[18] Samyam Rajbhandari, Jinsung Kim, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, Robert J. Harrison, and Ponnuswamy Sadayappan. 2016. On fusing recursive traversals of K-d trees. In *Proceedings of the 25th International Conference on Compiler Construction*. ACM, New York, 152–162. https://doi.org/10.1145/2892208.2892228

[19] Laith Sakka, Kirshanthan Sundararajah, and Milind Kulkarni. 2017. TreeFuser: a framework for analyzing and fusing general recursive tree traversals. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (October 2017), 76. https://doi.org/10.1145/3133900

[20] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is reasonable Coq.. In *In Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, New York, 14–27. https://doi.org/10.1145/3167092

[21] Akihiko Takano and Erik Meijer. 1995. Shortcut deforestation in calculational form. In *Conference on Functional Programming Languages and Computer Architecture*. ACM, New York, 306–313. https://doi.org/10.1145/224164.224221

[22] The Coq Development Team. Version 8.7.0. The Coq Proof Assistant. https://doi.org/10.5281/zenodo.1028037

[23] Philip Wadler. 1990. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science* 73, 2 (1990), 231–248. https://doi.org/10.1016/0304-3975(90)90147-A

[24] Yusheng Weijiang, Shruthi Balakrishna, Jianqiao Liu, and Milind Kulkarni. 2015. Tree dependence analysis. *ACM SIGPLAN Notices* 50, 6 (June 2015), 314–325. https://doi.org/10.1145/2813885.2737972